# Clojure & Incanter for Java Programmers

Ben Evans (**@kittylyst**)

One half of (**@java7developer**)

Slide Design by http://www.kerrykenneally.com

# Who is this guy anyway?

# Who is this guy anyway?

# Demo Details

http://192.168.1.73:3000/

Have a play, but don't deliberately try and break it

This is not a black hat / security demo

The (very simple) code is available from Github

My Github userid is kittylyst (same as Twitter)

# Today's Talk

# Clojure Overview

- Language Features Shopping List
  - JVM Hosted
  - Tight Integration with Java
  - Modern Concurrency
  - Novel Data Structures
  - Functional Programming
  - Rapid, Exploratory Programming

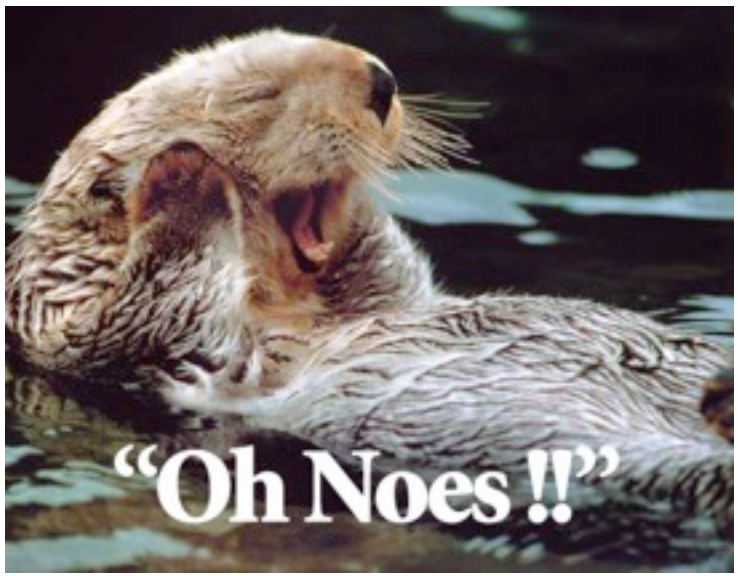
- Additional (possibly more surprising) Features
  - Advanced Code Transformation Capabilities
  - Dynamically typed, compiled language
  - Slim runtime

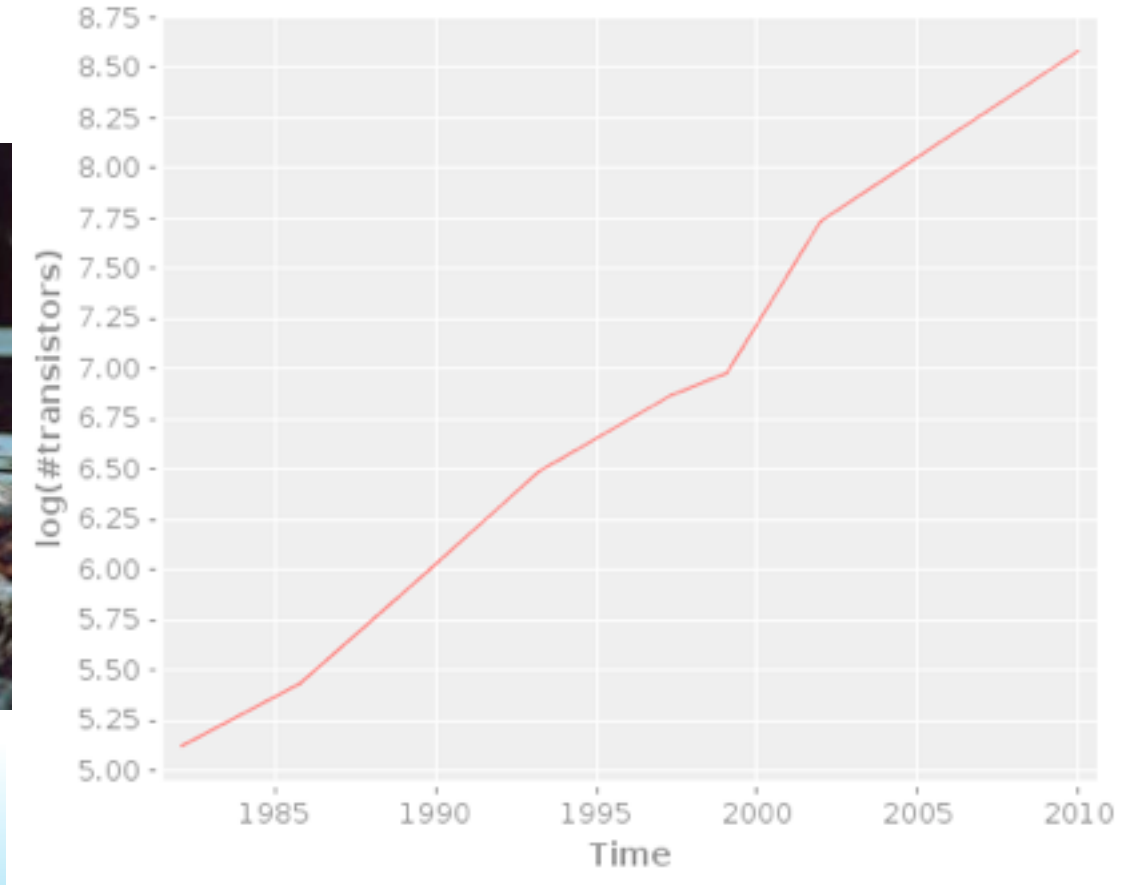
# JVM Hosted & Java Integration

- JVM is the most performant and flexible general purpose VM that exists
- Man-millennium of effort invested in HotSpot
- Why would you want to target anything else?

- Java has awesome tooling
- Great Libraries
- Huge ecosystem of products
- All can be leveraged easily if a non-Java language integrates well

# Moore's Law & Transistor Counts

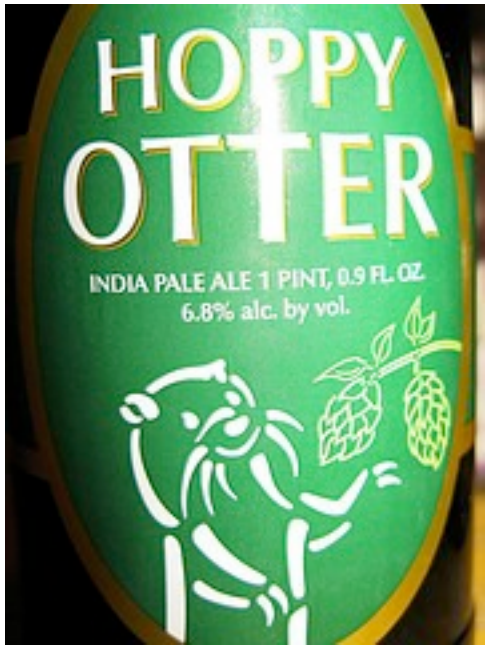


# Multithreaded Otters?

- Otters are a very good metaphor for concurrency
- Not just because they look a bit like threads (long and thin)
- Collaborative
- Competitive
- Sneaky
- Can hare off in opposite directions
- Capable of wreaking havoc if not contained

# Concurrent Java Code




- Based on `java.util.concurrent`
- Mutable state (objects) protected by locks
- Concurrent data structures (CHM, COWAL)
- Be careful of performance (esp COWAL)
- Synchronous multithreading - explicit synchronization
- `Executor`-based threadpools
- Asynch multithreading communicates using queue-like handoffs

# Concurrency - Overview

- Concurrency is key to the future of performant code (Moore's Law)
- Java's approach to concurrency looks increasingly dated
  - Incredible step forward in 90s, but we have ~15 years experience now
  - `java.util.concurrent` is a big help, but can only go so far
  - Java's syntax and semantics are millstones
- JMM is a low-level, flexible model
  - `Thread` is too low-level
  - The assembly language of concurrency
  - Need higher-level concurrency model


- Mutable state is hard
- Default sharing / visibility is usually wrong
- Locks can be hard to use correctly
- Need both synch & asynch state sharing

# Imagine a world...

- Collections were thread-safe by default
- “Objects” were immutable by default
- State was well encapsulated and not shared by default
- The runtime helped out the programmer more
- Thread wasn’t the default choice for unit of concurrent execution
- Copy-on-write was the basis for mutation of collections / synchronous multithreading
- Hand-off (via something queue-like) was the basis for asynchronous multithreading

# Wouldn't it be nice?

- Imagine all of that was built-in to the language
- But still built on top of the JVM
- Still based on the JMM view of memory
- What could such a language be?
- One thing's for sure - it's not Java
- We can't fix Java now, but we can learn from it (both good and bad points

# Clojure Syntax

## Clojure Syntax

# DON'T PANIC

# Keep Calm and Carry On

# Something you probably didn't expect:

# C Macros

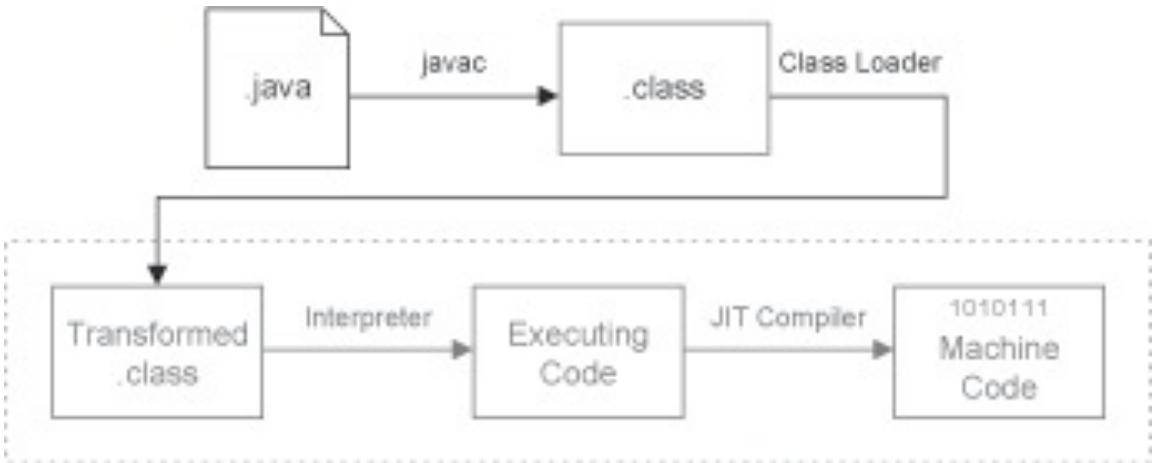
# Clojure Syntax - Code Transformation

```
#ifdef DEBUG
#define TRACE(m) {fprintf(stderr, "TRACE:%s\n", m);}
#else
#define TRACE(m) {}
#endif

#include <math.h>
#include <stdlib.h>

TRACE("Timelike boundary conditions OK!");
```

# Java Classloading



# Clojure Syntax - Code Transformation

Classloading can transform code as it is added to the running program.

E.g.

Aspect-Oriented Programming

Instrumenting Classloaders (e.g. EMMA)

Bytecode manipulation is cumbersome and difficult to get right. Tools like ASM make it easier.

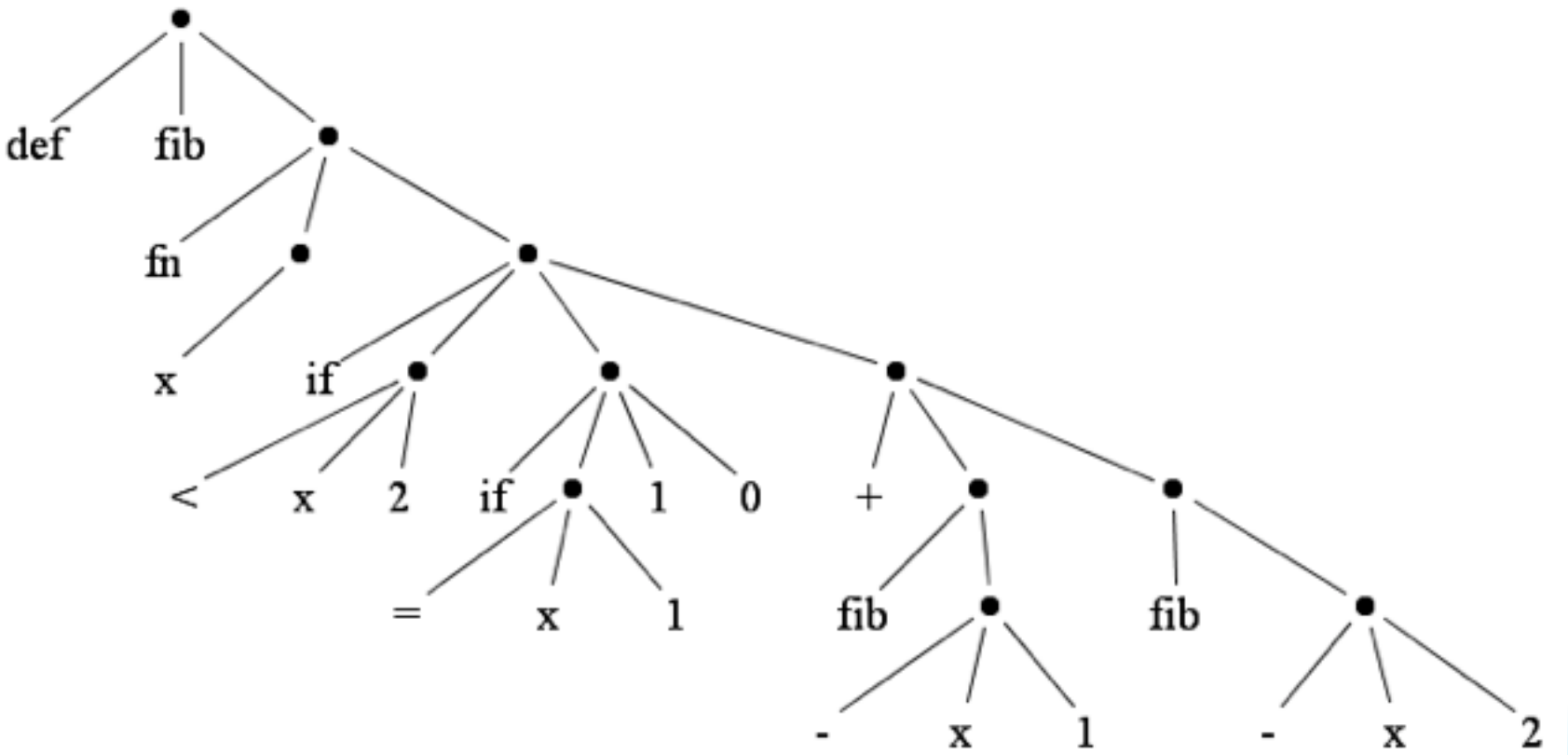
Is there a halfway house between C Macros and bytecode transformations?

# Abstract Syntax Trees

# Clojure Syntax - Code Transformation

```
public static int fib(int x) {
  if (x < 2) {
    if (x == 1) return 1;
    return 0;
  }
  return fib(x-1) + fib(x-2);
}
```

# Clojure Syntax - Code Transformation



# Clojure Syntax - Code Transformation

## Manipulating ASTs for fun and profit.

- Transforming code ASTs is *insanely* powerful.
- Not really a beginners topic, though.
- AST -> AST transforms are completely general way to express arbitary code transformations.
- Make a language syntax that looks like ASTs to start with.
- This simplifies the code transform syntax.
- Make the code transform syntax look like ASTs too (think “XML being transformed by XSLT”)
- That way, the programmer doesn’t need to learn two separate syntaxes

## 3 Simple rules to write down an AST

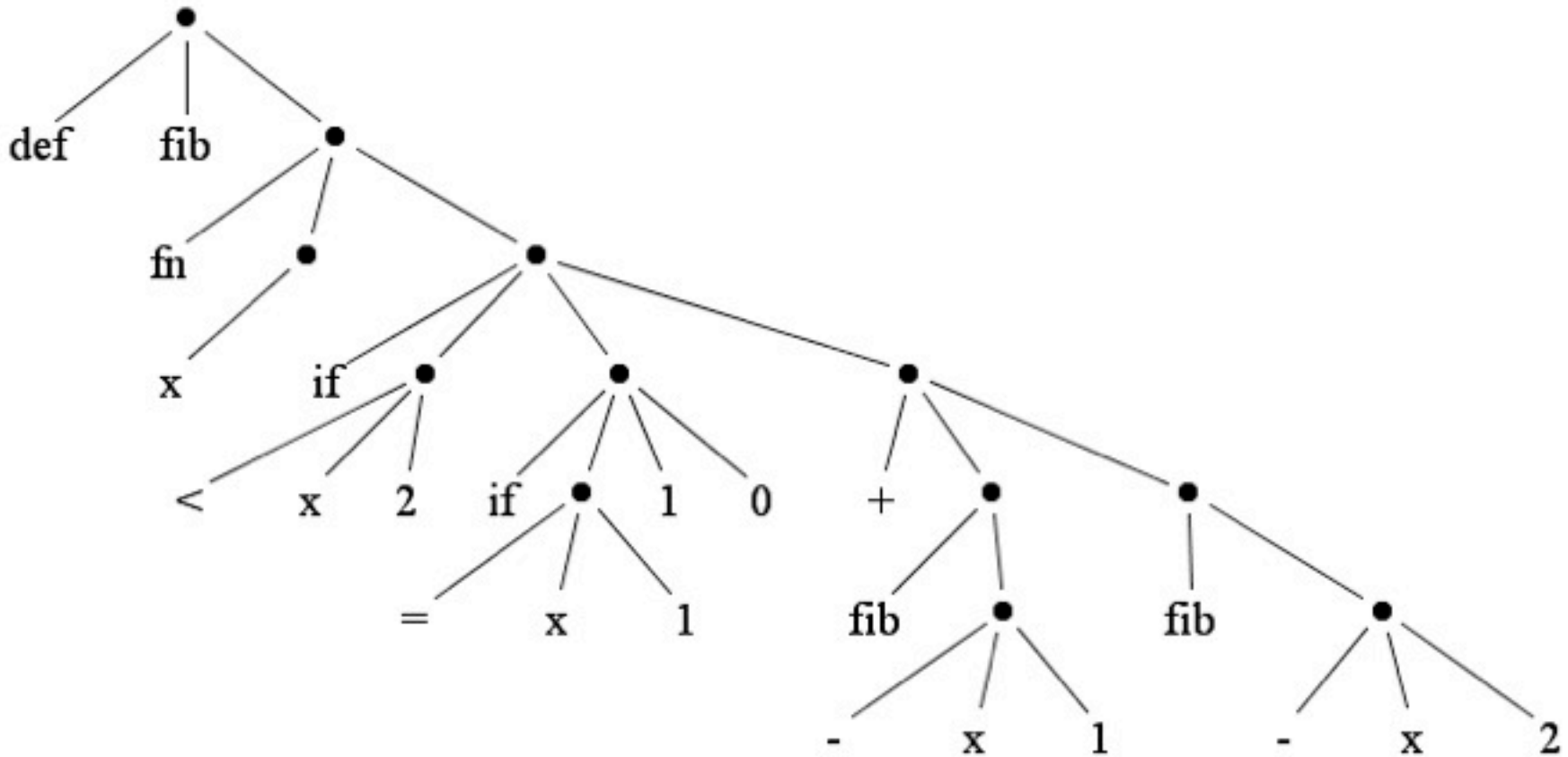
. = open bracket and move to left-most node of subtree

<token> = add to current position

<end of subtree> = close bracket, return up one level, and carry on

# Clojure Syntax - Code Transformation

```
(def fib (fn (x) (if (< x 2) (if (= x 1) 1 0) (+ (fib (- x 1)) (fib (- x 2))))))
```



# Clojure Syntax

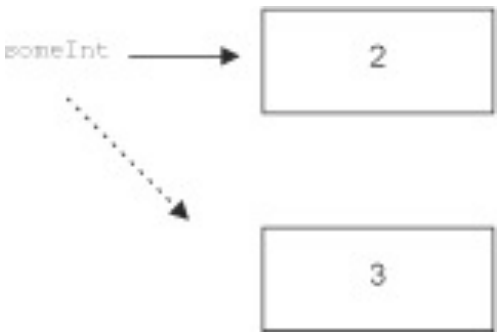
```
(defn fib [x]
  (if (< x 2)
      (if (= x 1) 1 0)
      (+ (fib (- x 1)) (fib (- x 2)))))
```

(...) is the Clojure list (think “LinkedList”)
[...] is a vector (think “ArrayList”)
{...} is a map (think “HashMap”)
x is a scalar variable (dynamically typed, remember)
No operators - arithmetic done by forms too (so no infix notation)
And, just for completeness ... Hello World:

```
(defn hello [] "Dydh da an Nor")
```

# Clojure - Immutability

Clojure is immutable by default




- Once created, values never change.
- (def) is used to bind names to values.
- (def) can be used to change the value a name points at.
- This is not the same as mutable state.
- (def) is always local to a thread, so anything which looks like mutation can't propagate.

# Clojure Syntax - Examples

```
   ; {} is an empty map (id -> img filename)
(def otter-pics {})

; defn makes a fn
; this one takes no params & returns random otter id
(defn random-otter [] (rand-nth (keys otter-pics)))

; returns next available id
(defn next-otter []
  (+ 1 (nth (max (let [otter-ids (keys otter-pics)]
    (if (nil? otter-ids)
        [0]
        otter-ids))) 0)))
```

# Clojure - Functions

```
(defn const1 [y] 1)

(defn ident [y] y)

(defn len-str [y] (.length (.toString y)) )
```

Now imagine passing these functions into other functions. E.g. “apply this function to every element of a list and give me back the resulting list” - which is the (map) form.

```
(map len-str ("a" "ab" "xyz" "st"))
```

# Clojure - Closures

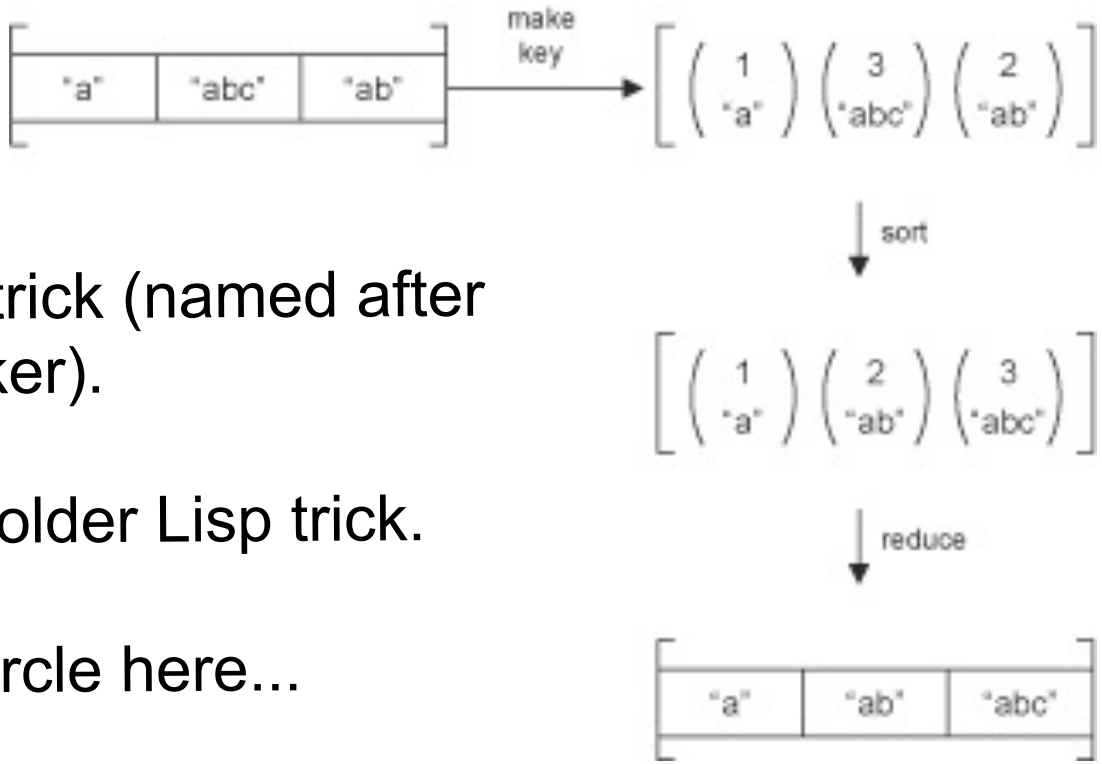
```
(defn adder [const-add] (fn [x] (+ const-add x)))

(def plus2 (adder 2))
```

`(adder)` is a function-making-function.
It takes in a constant and returns a new function.

The returned functions, such as `(plus2)`, are closures - because they "closes over" `const-add`.

# Clojure - Schwartzian Transform




This is an old Perl trick (named after a famous Perl hacker).

Based on an even older Lisp trick.

So has come full circle here...

# Clojure - Schwartzian Transform

```
(defn schwarz [x f]
    (map #(nth %1 0)
         (sort-by #(nth %1 1)
                  (map #(let [w %1] (list w (f w)))
                       x))))
```

Takes 2 arguments - a list (or more generally, “something that (map) operates on”) and a function which acts on elements of x.

Notice complete lack of static typing - even the function call is loosely defined.

#( ... ) syntax for inlined, anonymous functions

# Clojure - Functional Programming

Functions can be values (and put in vars, passed around etc)

The JVM handles a class as a basic unit of functionality (there's nothing smaller).

To the JVM a Clojure fn is just an instance of a class which has one method and no state (not quite true b/c of dynamic language nature, but a useful mental model).

Anonymous inner classes implementing an interface in Java are a similar concept.

There are many places where we see this in the Java APIs (think callback handlers, listeners, etc)

The term "SAM type" (for Single Abstract Method) is increasingly used - and will be important in Java 8.

# Clojure - Novel Data Structures

Notice also that we haven't really mentioned objects yet.

Clojure doesn't really "do" objects at the language level (although, of course, there are objects at the JVM level).

Instead, bundles of state are usually represented as maps, often using the `:key` syntax (called "keywords" in Clojure).

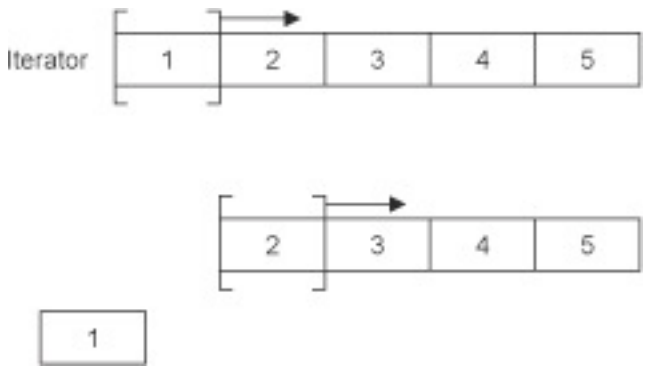
It's quite usual to just use one of Clojure's native data types (list, vector, map) as the basic data store.

Clojure doesn't use Java Collections. Let's see why.

# Clojure - Novel Data Structures

```
Collection<String> c = ...;

for (Iterator<String> it = c.iterator(); it.hasNext();) {
    String str = it.next();
    ...
}
```




Only 2 operations:

- Check to see whether the collection has any more elements in it
- Get the next element, and advance the iterator

# Clojure - Novel Data Structures

A better 2 operations:

- Get the head element `(first)`
- Get the sequence (called a seq) comprising the rest `(rest)`

This feels different - all references stay valid & nothing is mutating.

seqs act as “views” over collections, so can interoperate with Java collections code.

Multi-pass algorithms / work with retry is possible

lists are seqs, vectors are not.

Many libraries and Clojure abstractions are built on seqs

Nil-safe by design: `(first nil)` and `(rest nil)` are both `nil`.

# Clojure - Incanter

Incanter is a Clojure package for statistical computing.

Supports large (immutable) datasets, mathematical functions, graphing and Excel integration.

Similar ecosystem niche to R.

Use interactively or as libraries.

Large number of macros - e.g. ($=) for infix maths and vectorized operations

```
user> ($= [1 2 3] + 5)
(6 7 8)
```

# Clojure - Incanter

Interactive use of Incanter through the Clojure REPL is a very common use case.

Exploring large datasets is a great use of immutability.

In the demo, we'll explore the log files from the web app that you've been accessing through the talk.

# Clojure & Incanter Demo

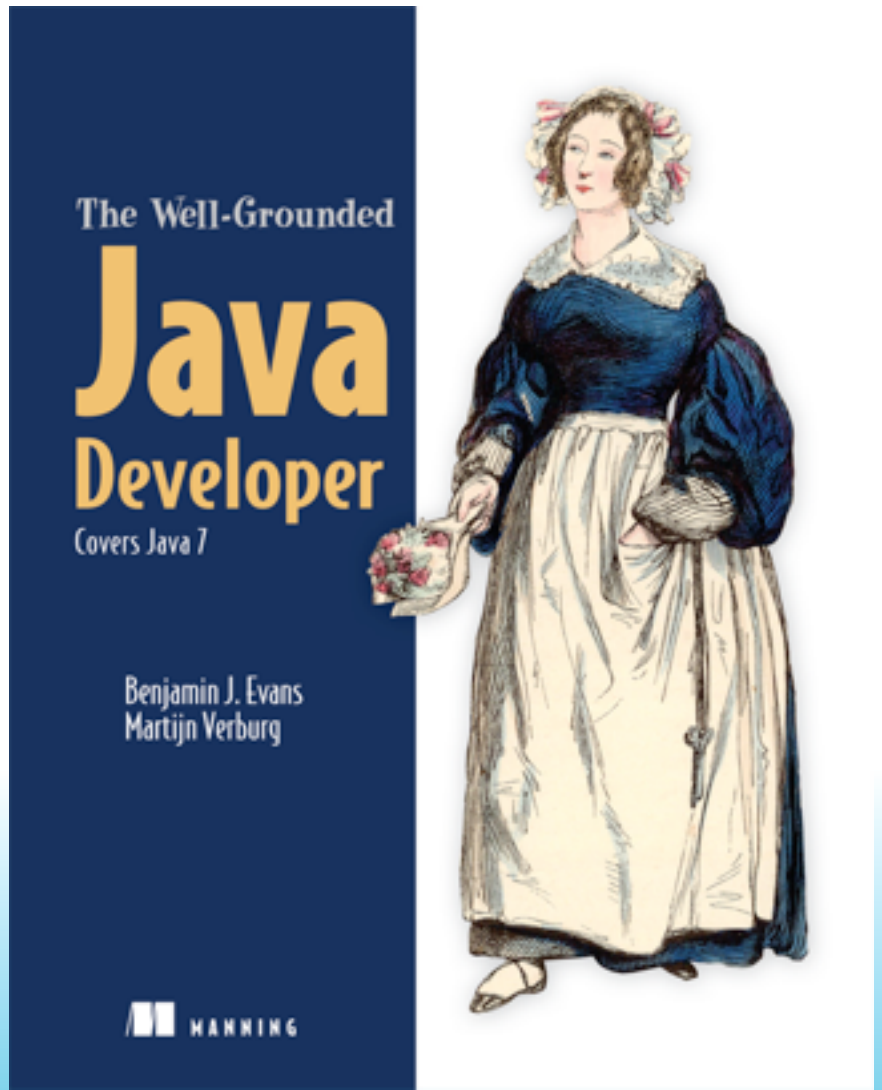
DEMO

# Clojure & Incanter: Stuff I Didn't Cover

- REPL-based TDD
- How to write Macros
- Protocols, multimethods, etc.
- Lazy Sequences
- Web development (Compojure, Ring, Hiccup, etc)
- Build systems (e.g. Leiningen)
- How Clojure implements new concurrency
- IDEs and productive working environments

- Fortunately, London has a thriving Clojure community, where you can learn all about these (and many, many other) subjects.

# Thanks for listening! @java7developer
# http://www.java7developer.com/



# Example - Compojure

```
; Set up the routes: URL -> fn mappings
; These are for the main pages & handlers
; for "Am I An Otter or Not?"
(defroutes main-routes
  (GET "/" [] (page-compare-otters))
  (GET ["/upvote/:id", :id #"[0-9]+" ] [id]
     (page-upvote-otter id))
  (GET "/upload" [] (page-start-upload-otter))

  ; This is the file uploader
  (mp/wrap-multipart-params
    (POST "/add_otter" req
       (str (upload-otter req) (page-start-upload-otter))))

  (route/resources "/")
  (route/not-found "Page not found"))
```